CEWES MSRC/PET TR/98-12

Comparing Middleware Support Systems for Collaborativc Visualization

by

Edward L. Peters
M. Pauline Baker

DoD HPC Modernization Program
Programming Environment and Training
CEWES MSRC

03h00598

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC 94-96-C0002
Nichols Research Corporation

Comparing Middleware Support Systems for Collaborative Visualization

Edward L. Peters
M. Pauline Baker
[elpeters | baker]@ncsa.uiuc.edu

Fall 1997

Habanero and Tango are two freely available packages designed to support the development of collaborative applications. We experimented with these middleware support systems to enable collaborative use of an existing tool for 3-dimensional visual data analysis. In each case, migrating our application presented challenges, but was ultimately successful. Here we report on that experience and compare and contrast these two middleware support systems in terms of their abilty to support our application.

1 Introduction

Increasingly, scientists conduct their research in teams that are geographically distributed throughout the nation or around the world. Teleconferencing is increasingly common to eliminate travel and the number of days away from the home institution. Many scientists have also come to rely on visualization as an integral part of the research process. To share the job of data analysis, scientists currently revert to sending hardcopy visualizations, following up express deliveries with telephone discussions in which they analyze the data. These scientist teams would be well served by on-line visualization tools that allow them to connect to their remote colleagues and share a data analysis session.

Collaborative visualization tools must consider the variety of environments in which researchers work. Even within a single team, some researchers might work on graphics workstations running Unix, while others use Windows-based PC's. The use of virtual reality for visualization is increasingly common – many researchers have access to devices such as the ImmersaDesk™, while a few have access to multi-screen projection environments such as the CAVE™. These systems[1] provide stereoscopic, immersive data analysis environments. Different graphics capabilities and different user interface libraries, obstacles in any cross-platform development effort, contribute to the difficulty of building collaborative visualization tools. Additionally, researchers network connections vary. Even as the number of high-speed connections increases, researchers on the road will also connect via phone or other low-speed lines to participate in shared data analysis sessions. Tools to support collaborative visualization must take into account these differences in graphics capability and connectivity.

Collaborative visualization tools must also support dialogue needed among the parties. Control over the visualization must be shared, with each party having an opportunity to manipulate parameters related to the visualization. Annotation capabilities, including the ability to point at or draw on the visualization, allow participants to make shared references to features shown in the visualization. It might also be necessary to provide audio or chat mechanisms. And in large collaborating groups, it might be useful to allow for subgroups to form.

Collaboration middleware systems are designed to support shared applications. Habanero and Tango are two freely available packages designed to support the development of collaborative applications. Habanero is a product of NCSA[2]. Tango comes from Northeast Parallel Architectures Consortium[3]. Both systems support reflecting a participant's activities to other members of the session. Habanero is entirely Java-based, while Tango supports both a Netscape plugin and a Java control applet.

In this paper, we report on our early experience in adapting an existing visualization application to run on the Habanero and Tango frameworks. To keep this experiment simple, we work with a single visualization application, and participants execute their copy of the application on relatively similar machines. The hardware mix includes 2 Octanes and 2 R4400 Indigo2 workstations, one with High Impact graphics and one with Extreme graphics. All the machines run IRIX 6.2 except for the Extreme machine, which runs IRIX 5.3.

Adding collaborative functionality as an "afterthought" to an existing application highlights the difficulties in building collaborative applications. We use these problems to construct a set of principles to guide the design of future collaborative applications.

2 Cbay

Cbay is a program for 3D visualization of flow and chemical data generated by computational simulations of phenomena in the Chesapeake Bay. *Cbay* allows the user to view data as colored slices, isosurfaces, and glyphs that relate multiple variables[4]. *Cbay* uses VTK to map the data to graphic form. VTK, freely available for both Unix and Windows machines, is a collection of C++ classes to read and manage scientific data, and to map data to graphics[5].

Cbay has a point-and-click user interface, written in TCL/Tk. This interface simply prints command strings to `stdout`. At startup, *Cbay* uses the Unix system call `popen` to the TCL/Tk interface. Subsequently, the main loop in *Cbay* polls this interface, reads new control strings, and executes. This simple design makes it easy to add other input modalities, such as a voice-recognition engine and a speech interface. A diagram of this setup is shown in Fig. 1.

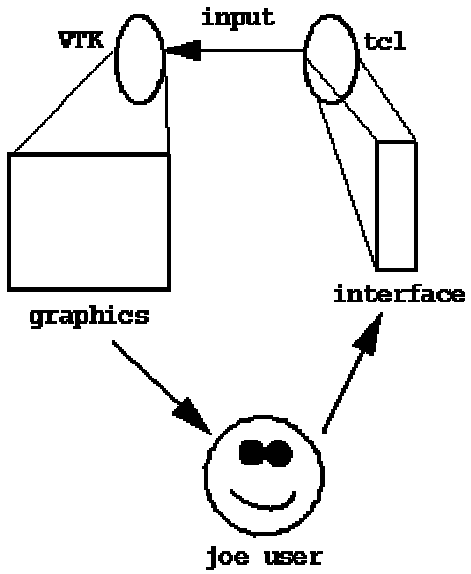

Fig. 1. The user interacts with the TCL/Tk interface, which passes strings to the graphics program.

Examples of the control messages are:

```
changeDay 0
ugridActor setRevealValue 0.00
ugridActor Animate Start
ugridActor Animate Stop
hogActor on
hogActor setScale 0
hogActor setScale 1
hogActor setScale 2
Navigate -1  0
Navigate -1  1
Navigate -1  3
Navigate -1  4
Navigate 0  6
Navigate 0  8
Navigate 0 0
Rotate Start
quit
```

These messages form a control protocol between the GUI and the main application, which performs rendering. Messages can be applied to the global application state, or to specific *actors*. Actors represent basic elements of the visualization, such as the wireframe grid, isosurfaces, slices, and so forth. The format of each message is generally `message [parameters]` for global messages such as Navigate and Rotate, and `actor message [parameters]` for actor-specific messages.

Two effects of this distributed control strategy are worth noting here. First, user input is centralized within the main application. The user interacts solely with the TCL interface, and commands from the TCL interface are handled by a fetch-and-execute cycle within the main application loop. This proved to make *Cbay* much easier to "Tango-ize" than to "Habanero-ize" (more on this later).

Second, while the VTK process can read from the TCL process, the reverse is not true. In the simple setup described here, the interface cannot be updated from within the main control loop. This is a problem if there are multiple sources of control over the application, which is the case in a collaborative scenario.

3 Cbay + Habanero = Cbay'nero

Habanero is a Java object-sharing framework. It handles networking and arbitration details to enable client applications to share state data and important events. It is distributed as a bundle of Java software including:

- A collaboration server (the `HabaneroServer`)
- A collaboration client (called simply `Habanero`)
- A number of demo applications and tools, ranging from time-wasters such a checkers applet and a collaborative calculator to useful tools such as an audio chat applet (not supported on SGIs) and a collaborative whiteboard.

Habanero works "out of the box". We found it easy to fire up the server and the client and test out some of the demos programs provided. Some of them are difficult to figure out without documentation, but the basic ideas come across loud and clear – this is a tool for developers of shared applications.

The first task in "Habanero-izing" Cbay was to rewrite the GUI as a Java application. This GUI would be distributed using Habanero, and would be responsible for starting the native graphics application. This is the reverse of the original control structure, where the graphics application launched the user interface. While the basic functionality was easy to implement, it took several iterations to get it "right". At the heart of the task was the Habanero event model.

3.1 Under the hood

This section assumes that the reader is familiar with Java. The two main components of the Habanero system, as mentioned above, are the `HabaneroServer` and `Habanero`, which is a graphical control panel for collaborants. Individual users launch applications from their control panel, which communicates through the `HabaneroServer` as shown in Fig. 2.

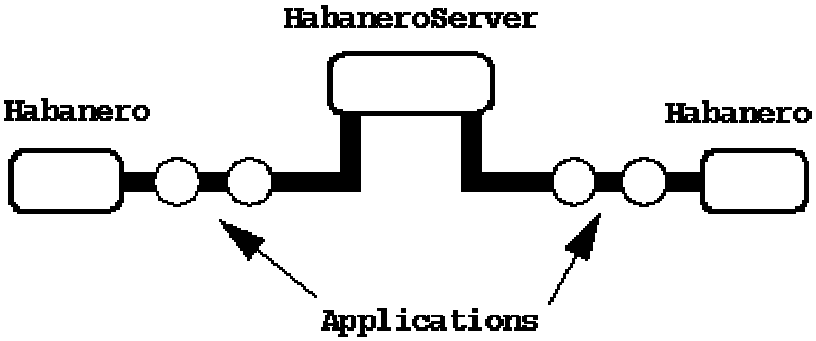

Fig. 2. The Habanero Server connects multiple Habanero applications.

Habanero's user environment is based on a "collaborating twins" approach: an application opened by one user is immediately opened for all other users. Individual events within this application can be shared or not, depending on the developer's decision. Since all applications are written in Java, they can be developed and distributed without regard to the target platform ("write once, run anywhere").

At first glance, Habanero's programming model might seem somewhat strange. Every collaborative application is started in a `ncsa.habanero.MirrorFrame`, a subclass of `java.awt.Frame`. Those AWT Events not handled by the local application are passed along to the Habanero framework and broadcast to all participants.

Events lose their context when broadcast to remote machines. This means that pressing a button on the local instance of the application will broadcast an event indicating a button was pressed, but can't indicate which button (since the button lives on the local machine only). The solution to this is to alter the parameters of the Event to indicate something about its context, then pass it Habanero. Pseudo code for the event handler would look something like this:

```
public boolean handleEvent (Event e) {
     if (e is a push on button 1) {
        e.arg = "Button 1";
        }
     else if (e is a scroll on scrollbar 1) {
        e.arg = "Scrollbar 1 has value" + scroll1.getValue();
        }
```

```
// This will forward the Event on to Habanero.
return false;
}
```

Note the capability to add arbitrary parameters to the Event on its way through the "first pass" event handling. In the code above, these are simply Strings; they can, however, be arbitrarily complex Objects, which implement the `ncsa.habanero.Marshallable` interface.

Also, not all Events must be passed on to Habanero. By returning `true` from an event handler, an application signals to Habanero that this Event does not need to be broadcast. This makes it easy to differentiate local and global (i.e. collaboration-wide) Events.

A broadcast Event is received from the Habanero framework in a special event handler called `doEvent`:

```
public boolean doEvent (Event e, Object arg) {
   // The argument is whatever object we assigned to e.arg
   String message = (String)arg;
   if (message.startsWith("Button 1")) {
      // Do something here
      }
   else if (message.startsWith("Scrollbar 1")) {
      // Do something here
      }
   // This event is over; it won't be forwarded any more
   return true;
   }
```

This "second pass" event handler is invoked on all machines, even the one that sent the event in the first place. Thus, first-pass event handling is usually minimal, and real changes to the program state take place in the call to `doEvent`. This layered event handling has two side effects:

1. Since all participants receive a broadcast Event, including the generator of the Event, all participants suffer a network delay (the time to transmit an Event from one client to a server and then to another client) in handling Events. This can even out timing delays between remote collaborants.

2. Once an event has been received, some work must be done to identify which GUI component(s) need to be updated. For instance, in selecting one of a group of list items, the appropriate item must be located and highlighted on all machines. Since standard GUI components generate rather than receive events, this makes code somewhat cumbersome.

One further point to mention is that Habanero messages are *only* generated by unhandled AWT Events. There is no general `send()` function in Habanero. To broadcast non-GUI messages to all participants, a new Event must be created and posted to the `MirrorFrame`.

3.2 Swimming upstream

The first approach to combining Habanero and *Cbay* was to retain as much of the original application structure as possible. A Java GUI was constructed which would generate string messages based on user input. These messages, in addition to being sent to the *Cbay* application, would be broadcast to other Habanero participants.

This approach made for an awkward event-handling model. The basic steps a GUI component went through to handle an Event were:

1. make necessary changes to state and generate String messages, encoding them with a unique user

ID

2. broadcast these messages by generating a fake AWT Event as described above
3. don't handle this new Event (return `false` so it will be broadcast by Habanero; notice the recursion in event handling here)
4. return `true` from the original event handling call to signify that the original Event has been handled

This was intended to enable the implementation of "following" or "over-the-shoulder" views, wherein one participant could shadow another simply by listening to his or her broadcast control messages. This scheme proved to be overly complicated since it introduced recursive event handling and messages from multiple sources. It was tricky to implement because it was a brute-force attack against the Habanero model of totally shared application state. The next design softened some of the corners.

3.3 Going with the flow

In the redesigned application model, each user had a `State` object, composed of parts like `NavState` and `DataState`, some of which were `Marshallable`. For example, the I/O state of the application was not shared, while visualization parameters like isosurface levels were shared. When a change was made to a shared data item, such as the transparency level of the wireframe grid, the corresponding portion of application state was broadcast to all participants. (It was added as an argument to the AWT Event, as described above.) If non-shared state was changed, no state object was broadcast. Each participant did his or her own translation into string messages.

This approach eliminated the need for fake Events and simplified event handling a great deal. The ability for participants to shadow one another was still available, albeit informally: nobody touches the interface but the person in the lead. Overall, this approach was more consistent with the Habanero philosophy of shared application state, which simplified things immensely.

3.4 Comments and future directions

The Habanero programming model is elegant, despite its initial complexity. Marshallable objects are easy to define and share, and message passing is an easy-to-understand extension of AWT event handling. In addition, Habanero allows the definition of an Arbitrator object, which acts on the central server to handle the routing of messages. The ability to define application-specific arbitration behavior is potentially quite powerful. Our first two versions of Cbay’nero did not take advantage of this, but it is in the works for future versions.

One of the main failings of Habanero was the lack of documentation. Our fully-developed Habanero application used exactly two Habanero classes: `ncsa.habanero.MirrorFrame` and the `ncsa.habanero.Marshallable` interface. Dozens of other classes and several entire packages remained unused – perhaps because they were unnecessary, but also because their function was not clear. Figuring out how to perform basic interaction with the collaborative environment (for example, querying for the names of all session participants) involved browsing through largely uncommented source code and javadoc-generated manual pages which contained no actual comments.

The other problem with using Habanero for collaborative visualization was its tight integration with Java and the AWT's event-handling model. Java itself currently provides no 3D graphics capability. This meant that interaction with a native graphics application was limited to using features currently available

in Java (mostly the `java.lang.Process` class), which in turn have their failings. Dependence on the AWT also meant a major overhaul of the Habanero system when the AWT changed in version 1.1.

4 Chesapeake + Tango = Chesapetango

Tango is a collaboratory environment focused on the World-Wide Web. Its basic infrastructure consists of a central server with multiple clients connected through web browser plugins. Tango presents APIs for Javascript, Java applets and applications, and C++ applications. At this writing, the API for Java applications remains only a specification.

The basic architecture is straightforward, though not intuitive. A server runs on some machine. Clients download a control applet, or CA (presumably from the server machine, but not necessarily) which connects through their web browser to the server. The control applet will then start applications as requested by the user. Local applications (LA) started from the CA will connect to the server through the browser, as will applets (A) running within the browser.

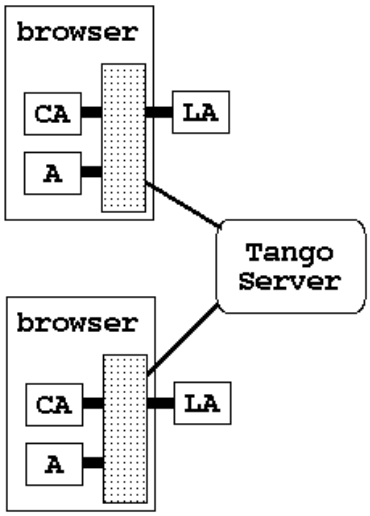

Fig. 3. The basic Tango architecture.

The Control Applet presents the main interface to the Tango system. It consists of a central control window containing information about collaborants, running sessions, and available applications. The CA and the browser plugin share information about where applications are located and which applications can "talk" to one another.

The basic software distribution from NPAC consists of the following:

- The server (implemented in Java); and,
- A Netscape plugin (implemented mostly in Java, with some fraction in C++).

Conspicuously absent from these distributions is the CA. This applet, along with its configuration files, is maintained by the developers of Tango. The Tango team at NPAC supplied a version of the CA (.class and configuration files only), which allowed for running the whole system locally. This improved speed and also helped in understanding of the system.

4.1 The Tango API(s)

Tango advertises APIs for Javascript, Java applets and applications, and C++ applications. This is strikingly different from Habanero's single-language approach, and is more promising for developers of graphical applications. A sample Java applet is available which implements a collaborative web browser. The Java application API was actually never implemented in version 1 of the Tango software. This project focused on integrating a C++ application (*Cbay*) into the framework. Comments here are restricted to that particular API.

The Tango API for C++ applications is refreshingly simple. It is included here in its entirety, partly for reference and partly to underscore its brevity.

```
#define CONTROL 30
#define DATA    31

struct Event {
   int type;
   int  len;
   char *data;
};

int registerApp(int AID, int port);
void sendEvent(int sockfd, struct Event *e);
struct Event *receiveEvent(int sockfd);
struct Event *createEvent(int type, int len, char *data);
int getType(struct Event *e);
int getLen(struct Event *e);
char *getData(struct Event *e);
void destroyEvent(struct Event *e);
```

The steps of initiating a Tango C++ application are as follows:

1. From the Tango control panel, a button is pushed which signals the CA to begin a local program;
2. This local program is started with command-line arguments to indicate the application ID of the current application and the port number of the local Tango daemon;
3. The local program uses these parameters in a `registerApp` call, which returns an open socket descriptor;
4. This socket is used in calls to `sendEvent` and `receiveEvent`.

While simple, this API is not easy to use without further knowledge of the exact semantics of each of the routines. In particular, the part about the command-line arguments was not immediately obvious from the documentation, and it was only through trial and error that it became apparent that `receiveEvent` was a blocking read. The lack of sample code for the C++ API was a major hindrance. The Tango team graciously yielded some demo code that served to illustrate the basics of the API.

In the end, the path of least resistance included writing a new class called `TangoEvent`, which served as a wrapper around the Tango API provided by NPAC. Difficulties with the `sendEvent` class distributed with Tango inspired some local re-implementation of that class to get everything working.

4.2 The golden spike

Once the basic control structure and Tango API was understood and the local wrapper classes working, it was a twenty-minute effort to shoehorn *Cbay* into the Tango framework. This was mainly because of the centralized control structure in the original application. Instead of:

```
1. fetch command from TCL interface;
2. if there is a command, do it;
```

the event handling routine was more like:

1. fetch command from TCL interface;
2. if there is a command, do it *and send it to Tango;*
3. *fetch command from Tango;*
4. *if there is a command, do it.*

Note that, unlike Habanero, Tango does not echo events back to their originator. The C++ main program still created the same TCL GUI, and the same string messages were sent out to other Tango participants.

4.3 Comments

The Tango system has many positive features for designers of collaborative visualization systems. One of its strongest points is the ability to use C++. This allows the easy integration of native, optimized graphics software. In addition, there are a wide set of mature tools for application development in C++ that do not exist in Java (of particular interest to us, for example, are well-developed system calls to start and interact with other processes).

Another advantage is the ability to connect heterogeneous applications to a collaboration session. Each participant's plugin reads data from a configuration file. In that file, a participant can indicate a preference for a particular application to be used when joining a session of a particular "type". It might be that one participant is in a CAVE with a highly immersive visualization application, a second is at a graphics workstation running a 3D visualization tool, and a third is at running a 2D visualization tool on a PC. If all three applications understand the same event API, and each application can respond in some meaningful way to these events, then the three participants could share their data analysis session, supported by the Tango framework. This would allow an exploration of the issues involved in collaborative visualization between users with radically different displays.

Finally, the message-passing architecture was very simple to understand. There is no substantial overhead required to figure out the Tango API.

For this project, the message-passing architecture was particularly well suited to our application. As described above, the original *Cbay* application responds to a set of control strings generated by the TCL/Tk user interface. This made Tango, with its message-passing scheme, a particularly good match for supporting our application.

All that glitters is not gold, however; getting started with the Tango environment was difficult. As mentioned earlier, the Tango C++ API is refreshingly simple, but contains some hidden details that cannot be overlooked. The three most important points we discovered are:

- The `registerApp` call returns a socket descriptor, which is a connection to the local daemon. Some example code received from NPAC had code for this call but it didn't work.

- The `receiveEvent` call is blocking; a call to `ioctl()` (or some similar system call) on the socket descriptor mentioned above will reveal if there is anything waiting to be read.

- The `sendEvent` call simply sends two 4-byte integers and a character buffer. The first integer is the CONTROL/DATA indicator, and the second is the length of the buffer. A `sendEvent` routine had to be implemented.

These are not intricate problems, and can be solved with the application of a little bit of guesswork and some UNIX networking know-how; however, stronger documentation and more commented example code could reduce the learning curve significantly.

One other important factor about Tango is the complexity of the system itself. In the combination of Netscape, Java, and TCP/IP networks, it is possible for strange errors to occur. Most of them can be avoided by careful attention to the compatibility of different versions of software, but when problems do arise, they can be difficult to diagnose.

Tango is somewhat cumbersome when it comes to debugging applications. Error output from running applications is presented in Netscape popup windows. These windows can be buffered and, if diagnostic output is voluminous, it will be truncated, perhaps inconveniently.

5 Conclusions

In these attempts to “collaboratize” an existing visualization application, certain aspects of the task proved to be particularly difficult. These were independent of the collaboration middleware used, and had more to do with basic aspects of application design: defining application state, maintaining coherence between the application and its interface, and using time-based rather than frame-based animation.

The first problem, and perhaps the most fundamental, was the question of defining application state in such a way that parts of it could be shared. Ideally, this could be segmented off in some kind of a shared distributed object. In practice, coherence among collaborants was maintained by passing messages back and forth. For *Cbay*, the content and format of those messages was predefined (since application state already changed in response to string messages). The question still remained, however, of how best to partition application state into shared and private data.

Using Habanero, it was easiest to represent the application state as a composite Java object, parts of which could be made marshallable and shared. For Chesapetango, the solution was simply to share all application states by sharing all control messages. A more cautious approach would involve parsing control messages to watch for state updates.

This brings up the problem of interface coherence. As described above, the GUI was designed to be "one way" – it generated messages and sent them on to the application. The application state changed only in response to user input received through these messages, and thus the GUI was synchronized with the display. However, in collaborative scenarios, there are multiple sources of input. The user interface needs to keep up not only with the actions of a single user, but of all participants in the collaboration.

In Habanero, this can be difficult. The partial solution in Cbay’nero was to break the interface into many separate dialogs which would update themselves from application state when opening or closing. This was mostly for simplicity and managability, so I didn't have to worry about finding which GUI component to update in response to a message. This problem remains unsolved in the Tango-*Cbay* effort, and the interface viewed by any one participant does not necessarily match the visual.

Another coherence problem arises from using frame-based animation on different machines. Our lab consists of multiple SGI workstations with widely different graphics capabilities, and thus different frame rates. The user on one machine might see an object spinning ten times faster than the user on another machine. Time-based animation (either based on wall-clock time or some kind of broadcast simulation

time) would mitigate these coherence problems. Some Habanero applications have included an Arbitrator which sends out regular clock ticks; this would be the responsibility of one of the collaborants in Tango.

Finally, even the use of time-based animation would not eliminate the usefulness of some kind of a base or "reset" state. If, for some reason, two collaborants got "out of synch" with one another, they could return to a common state and proceed. Extending this notion to allow synchronization at an arbitrarily specified state would also allow some "do you see what I see?" functionality. This is strictly a matter of application design.

Both Habanero and Tango open the door to interesting future work exploring different topics in collaborative visualization systems. For instance, Habanero allows an application-specific central Arbitrator, which can function as a gateway to another system (previous projects have included collaborative telnet sessions and Habanero/NICE connections). A collaborative session with a remote visualization generator could take advantage of various tools in Habanero, like a collaborative whiteboard for marking up snapshots, or a collaborative VRML browser for interaction with 3D data.

The current versions of both Habanero and Tango rely on the presence of the basic control panel, from which all applications must be launched. Neither package supports the concept of a user-defined client interface. Later releases of Habanero will support operation without the control panel. It is not clear whether later versions of Tango will do so as well. Removing this restriction will be particularly useful in applications for environments such as the CAVE, where effective presentation of the graphics depends on having complete use of the console, uncluttered by unnecessary control panels.

In summary, both Habanero and Tango are suitable to solving different problems in designing collaborative visualization applications. The choice of which package to use depends on the characteristics of the application under development. In our case, where our application was written in C++ and already responded to a well-defined set of control strings, Tango proved to be the more suitable middleware support system.

References

[1] http://www.ncsa.uiuc.edu/Vis/facilities.html

[2] http://www.ncsa.uiuc.edu/SDG/Software/Habanero

[3] http://www.npac.syr.edu/tango

[4] http://www.ncsa.uiuc.edu/Vis/Projects/Chesapeake

[5] Schroder, W., Visualization ToolKit (2nd Edition), Prentice-Hall, 1998.